Aula 7 - 25 de mar?o

March 24, 2020

1 Aula 25/03

1.1 Ainda recursão

Nessa aula vamos continuar exercitando o que já vimos nas aulas passadas, isso é, escrever testes antes de escrever o código e continuar explorando a recursão.

Para começar, vamos fazer um exercício de aquecimento, isso é o cálculo de potência de um número. Nesse caso específico, queremos elevar um número x a uma potência inteira positiva n.

É de se esperar que a assinatura da função seja algo como **pot(x, n)**, logo vamos começar com os testes:

```
In [ ]: function testapot()


        end
```

Agora de posse da função, podemos pensar em como resolver o problema. Vamos pensar no caso de 2^4, podemos escrever isso como $2 * 2^3$, sendo que 2^3 pode ser escrito como $2 * 2^2$, assim por diante até chegarmos em $2 = 2 * 2^0$.

```
In [ ]: function pot(x, n)

        end
```

Vamos agora para um exercício um pouco mais sofisticado, o cálculo da raiz quadrada pelo método de Newton.

Basicamente queremos achar um valor $y \geq 0$ que corresponda a $\sqrt{x}$ ou seja $y * y = x$

Vamos ver como se calcula por exemplo a raiz de 2.

Vamos começar com um chute inicial 1 (ou seja metade de 2).

chute	quociente
1	2/1 = 2
1.5	2/1.5 = 1.3333
1.4167	2/1.4167 = 1.4118
1.4142	...

Vamos ver como calcular a raiz de 3, sempre lembrando que um bom chute inicial é a metade

do valor

De alguma forma, vimos que com o número para o qual queremos encontrar a raiz e com o chute inicial, podemos encontrar o próximo palpite, estamos quase lá, mas vamos pensar agora no teste.

Como estamos pensando em números com ponto flutuante é bom lembrar que agora a comparação pode não ser exata.

Para não ter que pensar mais em sinal, vale lembrar que em Julia temos a função abs(x) que devolve o módulo de x

```
In [8]: println(abs(-10))
```

```
10
```

Agora sim, podemos pensar nos testes

```
In [ ]: function quaseigual(a, b)
          erro = 0.0001
        end

        function testaraiz()
        end
```

Agora sim, vamos fazer a função **raiz(x)** propriamente dita, como para a recursão, será necesário passar chute, nada mais natural do que ter a função abaixo

```
In [ ]: function raiz(x)
            return raizrec(x, x/2.0)
        end

        function raizrec(x, guess)
        end
```

Vamos brincar um pouco agora com funções caóticas :), isso é, funções, que conforme o comportamento de uma constante k, apresentam resultados que podem convergir ou não. Isso é, a cada passo, quero saber o valor do próximo ponto aplicando a função novamente, isso é: $x_1 = f(x_0), x_2 = f(x_1), \ldots x_n = f(x_{n-1})$.

A função f é extremamente simples: $f(x_{i+1}) = x_i * (1 - x_i) * k$.

```
In [19]: function f(x)
             k = 3.7
             return x * (1.0 - x) * k
         end

         function imprimeN(n, x)
             if n <= 0
               println(x)
             else
               println(x)
```

```
            imprimeN(n - 1, f(x))
        end
    end
```

```
Out[19]: imprimeN (generic function with 1 method)
```

Tentativas iniciais, $k = 2.1, 2.5, 2.8, 3.1$, mas a coisa fica interessante para $k = 3.7$, por quê?

```
In [20]: imprimeN(20, 0.5)
```

```
0.5
0.925
0.2566874999999999
0.7059564011718747
0.7680532550204202
0.659145574149943
0.8312889390453945
0.5189162638048546
0.9236760473655611
0.26084484548817094
0.7133778046605658
0.7565386761694786
0.6814952582280823
0.8031200435906705
0.5850374849422831
0.8982439167723566
0.33818659618910557
0.82812076268439
0.5266460308530325
0.9223729594471832
0.264924007572964
```

A informação sobre funções caóticas foi obtida em: http://mathforum.org/library/drmath/view/51947.html